CZ4055 : Cyber Physical System Security

Project Report for Correlation Power Analysis

Prepared by
Lim Jing Qiang (U1722144E)
Koh Chin Woon (U1721286F)
Oon Zi Hui (U1722934E)

Lab Group CS4
Group work

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
NANYANG TECHNOLOGICAL UNIVERSITY

Content Page

1. Introduction

Electronic devices such as computers and mobile phones are widely used in our daily life, providing convenience to people as various activities such as bank transactions can be done on these devices. Hence, there has been an increasing concern on security risks of these electronic devices as attackers would try to obtain confidential data from it. The attacks to the electronic devices can either be active or passive. This report will be focusing on the Correlation Power Analysis algorithm and the countermeasures of power analysis attack, which is a form of passive and non-invasive side-channel attacks.

2. Implementation

The implementation for this project was split into two parts:

a) Correlation Power Analysis
b) Data visualization

After careful consideration, our group has decided to implement the Correlation Power Analysis using Java as it is statically typed and compiled, unlike Python, where it is dynamic and interpreted. The implementation will also be multithreaded to speed the analysis up.

However, the data visualization will be done using Python as we will be utilizing the libraries for plotting and analyzing graphs.

Correlation Power Analysis

The pseudo code to perform the Correlation Power Analysis is :

```
Initialize the actual trace matrix
   For each of the byte in key length{
        Initialize the model matrix for each byte
        Create the correlation matrix for each byte
        Find the key for that byte from the correlation matrix
        }
```

Below are the detailed description of each of the steps.

Model Trace Matrix Initialization

The model trace matrix using a 2D Array with the size of [*a*][*b*] where :

a is the number of sample traces

b is the number of possible values for 1 byte of the key ⇒ $2^8 = 256$

The model matrix is then initialized with values of *HammingWeight(SBOX[*$X_i \oplus K_j$*])* where:

HammingWeight is a function that calculates the number of binary '1's in a number

SBOX is an array that contains substitution values for a given index.

X_i is *i*th byte of the plaintext

K_j is the number of possible values for 1 byte of the key, where *j* ranges from [0 to 255]

The array for Sbox can be seen in Figure 2.1.1 and the implementation for initializing the model trace matrix can be seen in Figure 2.1.2. For initializing the model trace matrix, loop unrolling has been used to speed up the process.

```
public static int[] Sbox = {
        0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76,
        0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4, 0x72, 0xC0,
        0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8, 0x31, 0x15,
        0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2, 0xEB, 0x27, 0xB2, 0x75,
        0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3, 0x2F, 0x84,
        0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C, 0x58, 0xCF,
        0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0x7F, 0x50, 0x3C, 0x9F, 0xA8,
        0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3, 0xD2,
        0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D, 0x64, 0x5D, 0x19, 0x73,
        0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8, 0x14, 0xDE, 0x5E, 0x0B, 0xDB,
        0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC, 0x62, 0x91, 0x95, 0xE4, 0x79,
        0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A, 0xAE, 0x08,
        0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0xBD, 0x8B, 0x8A,
        0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0xB9, 0x86, 0xC1, 0x1D, 0x9E,
        0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55, 0x28, 0xDF,
        0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0x0F, 0xB0, 0x54, 0xBB, 0x16
};
```

Figure 2.1.1 SBox values

```
public void initModelMatrix(int byteNo) {
    modelMatrix = new int[powerTrace.size()][Analyzer.keyHyp.length];

    for(int i = 0 ; i<powerTrace.size();i++) {
        String frontByte = "0x"+powerTrace.get(i).getPlainText().substring(2*(byteNo-1),2*byteNo);
        for(int j = 0; j<Analyzer.keyHyp.length;j+=8) {
        //Loop unrolling....
            modelMatrix[i][j] = Integer.bitCount(Analyzer.Sbox[Integer.decode(frontByte) ^ Analyzer.keyHyp[j]]);
            modelMatrix[i][j+1] = Integer.bitCount(Analyzer.Sbox[Integer.decode(frontByte) ^ Analyzer.keyHyp[j+1]]);
            modelMatrix[i][j+2] = Integer.bitCount(Analyzer.Sbox[Integer.decode(frontByte) ^ Analyzer.keyHyp[j+2]]);
            modelMatrix[i][j+3] = Integer.bitCount(Analyzer.Sbox[Integer.decode(frontByte) ^ Analyzer.keyHyp[j+3]]);
            modelMatrix[i][j+4] = Integer.bitCount(Analyzer.Sbox[Integer.decode(frontByte) ^ Analyzer.keyHyp[j+4]]);
            modelMatrix[i][j+5] = Integer.bitCount(Analyzer.Sbox[Integer.decode(frontByte) ^ Analyzer.keyHyp[j+5]]);
            modelMatrix[i][j+6] = Integer.bitCount(Analyzer.Sbox[Integer.decode(frontByte) ^ Analyzer.keyHyp[j+6]]);
            modelMatrix[i][j+7] = Integer.bitCount(Analyzer.Sbox[Integer.decode(frontByte) ^ Analyzer.keyHyp[j+7]]);
        }
    }
}
```

Figure 2.1.2 Function to initialize the model trace matrix

PowerTrace Instantiation

Before the actual trace matrix is created, *PowerTrace* objects will be created by reading the collected trace .csv file. The *PowerTrace* object has the structure of :

{
int number_of_sample_points; //2500 sample points
double[] traces ; //Voltage of each sample point
String plain_text; //Plain text
}

The *PowerTrace* objects will then be stored in an ArrayList upon instantiation.
The implementation of reading the csv file and instantiation of *PowerTrace* objects can be seen in Figure 2.1.3.

```
public void loadPowerTraceFromFile(String csvFileName) {
    try (BufferedReader br = new BufferedReader(new FileReader(csvFileName))) {
        String line;
        try {
            while ((line = br.readLine()) != null) {
                String[] values = line.split(",");
                powerTrace.add(new PowerTrace(values));

            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    } catch (FileNotFoundException e1) {
        e1.printStackTrace();
    } catch (IOException e1) {
        e1.printStackTrace();
    }

}
```

Figure 2.1.3 Reading of .csv file and instantiation of *PowerTrace* objects

Calculating correlation

The calculation is done by utilizing Pearson's Correlation Coefficient equation to find out how correlated the two matrices are (Model trace matrix vs Actual trace matrix). The coefficient will range from -1 to 1, with 1 being the most correlated and -1 being the least correlated.

The Pearson's Correlation Coefficient can be calculated using the formula below :

$$r = \frac{n(\sum xy) - (\sum x)(\sum y)}{\sqrt{[\, n\sum x^2 - (\sum x)^2 \,][\, n\sum y^2 - (\sum y)^2 \,]}}$$

Where:

n is the size of the array
$\sum x$ is the summation of the elements in array X
$\sum y$ is the summation of the elements in array Y
$\sum x^2$ is the summation of the elements squared in array X
$\sum y^2$ is the summation of the elements squared in array Y
$(\sum x)^2$ is the summation of the elements in array X squared
$(\sum y)^2$ is the summation of the elements in array Y squared

The code equivalent implementation can be seen in Figure 2.1.4.

```
public double correlation(double[] arrayX, double[] arrayY, int size) {
    double sum_x = 0.0;
    double sum_y = 0.0;
    double sum_xy = 0.0;
    double sum_xx = 0.0;
    double sum_yy = 0.0;
    double x = 0.0;
    double y = 0.0;

    for (int i = 0; i < size; i++)
    {
       x = arrayX[i];
       y = arrayY[i];
   // sum of elements of array X.
      sum_x += x;
   // sum of elements of array Y.
      sum_y += y;
   // sum of X[i] * Y[i].
      sum_xy += x * y;

   // sum of square of array elements.
      sum_xx += x * x;
      sum_yy += y * y;
    }

    return  (double)((double)size * sum_xy - sum_x * sum_y)
           / Math.sqrt((double)((double)size * sum_xx - sum_x * sum_x)
                   * (size * sum_yy - sum_y * sum_y));
}
```

Figure 2.1.4 Pearson's Correlation Coefficient calculation

Creating the correlation matrix

The correlation matrix will contain correlation coefficient of the two matrices are (Model trace matrix vs Actual trace matrix).

The correlation matrix can be created using a 2D array with the size of [*a*][*b*] where :

a is the number of possible values for 1 byte of the key ⇒ $2^8 = 256$

b is the number of sample points in 1 trace. ⇒ 2500

The implementation of the this function can be found in the powerpoint slides provided. Figure 2.1.5 will show the code implementation.

```
public void createCorrelationMatrix() {
    corMatrix = new double[Analyzer.keyHyp.length][powerTrace.get(0).getTotalNoOfSamplePoints()];
    double x[] = new double[powerTrace.size()];
    double y[] = new double[powerTrace.size()];

    for(int count = 0; count<corMatrix.length;count++) {
        for(int j = 0; j<powerTrace.size();j++)
            y[j] = (double) modelMatrix[j][count]/256.0;

        for(int i = 0 ;i<powerTrace.get(0).getTotalNoOfSamplePoints();i++) {
            for(int j = 0; j<powerTrace.size();j++) {
                x[j] = powerTrace.get(j).getTraceWithIndex(i);
            }
            corMatrix[count][i] = correlation(x,y,x.length);
        }
    }
}
```

Figure 2.1.5 Creating the Correlation Matrix

Data Logging, Key Retrieval and Bits Recovered

In order to log down data that is required for this project, the number of traces is varied in an ascending order starting from 10. The total number of traces our group has collected is 1500.

The data we are interested to log down is the maximum value of each row of the correlation matrix per number of traces. However, we are also logging down the minimum value of each row to create show that the most of correlation coefficient will eventually converge(except the correct key) as the number of traces increases.

As for the key retrieval, we are interested in the location of the highest correlation value in the correlation matrices of each key byte.

Once the whole secret key has been recovered, the number of bits recovered can be calculated by converting the actual key and recovered key into binary and count the number of matching bits between the 2 set of keys. The number of bits recovered per number of traces is also being logged down.

The implementation of the correlation coefficient data logging can be seen in Figure 2.1.6.
The implementation of bits recovered calculation can be seen in Figure 2.1.7.

```
public String findKeyIndex() {
    int index = -1;
    String dataPlot= "";
    String dataPlot100Traces = "";
    double largest = -1.0;

    for (int i = 0; i < Analyzer.keyHyp.length; i++) {
        double rowLargest = -1.0;
        double rowSmallest = 1.0;
      for (int j = 0; j < powerTrace.get(0).getTotalNoOfSamplePoints(); j++) {
        if (corMatrix[i][j] > largest)
        {   //For analysis
            largest = corMatrix[i][j];
            index = i;
        }
        if(corMatrix[i][j]>rowLargest)//For logging down purpose
            rowLargest = corMatrix[i][j];
        if(corMatrix[i][j]<rowSmallest)//For logging down purpose
            rowSmallest = corMatrix[i][j];
      }
     dataPlot +=rowLargest+","+rowSmallest+",";
     dataPlot100Traces +=rowLargest+",";
    }

    if(dataPlot.length()>0) {//Store in ArrayList first before writing to csv
        dataPlot = dataPlot.substring(0,dataPlot.length()-1);
        dataToWrite.add(powerTrace.size()+","+dataPlot);
        dataPlot="";
    }
    if(dataPlot100Traces.length()>0 && powerTrace.size()== maxTraces) {//Store in ArrayList first before writing to csv
        dataPlot100Traces = dataPlot100Traces.substring(0,dataPlot100Traces.length()-1);
        dataToWrite_100Traces.add(powerTrace.size()+","+dataPlot100Traces);
        dataPlot100Traces="";
    }
    return  String.format("%02X", (0xFF & index));
}
```

Figure 2.1.6 Data logging of Max/Min Correlation Coefficient

```
public int bitsRecovered(String actualKey,String recoveredKey) {
    int correctBits = 0;
    String bin_actual = hexToBin2(actualKey.replaceAll(" ",""));
    String bin_recovered = hexToBin2(recoveredKey.replaceAll(" ",""));
    for(int i = 0; i<bin_actual.length();i++) {
            if(bin_actual.charAt(i)==bin_recovered.charAt(i))
                correctBits++;
    }
    return correctBits;
}
```

Figure 2.1.7 Calculate number of bits recovered

Sample output

The Java application will create *Graph.csv*, *MinMaxCorMatrix1.csv, MinMaxCorMatrix2.csv* …. till *MinMaxCorMatrix16.csv*.Graph.csv will contain the data for number of traces vs bits recovered. *MinMaxCorMatrix**1-16.csv*** will contain the max and min correlation values for each of the key bytes.

A sample output of the console can be seen in Figure 2.1.8.

```
Retrieving secret key using 10 number of traces...
Thread is analyzing key byte number 1
Thread is analyzing key byte number 2
Thread is analyzing key byte number 5
Thread is analyzing key byte number 4
Thread is analyzing key byte number 3
Thread is analyzing key byte number 6
Thread is analyzing key byte number 10
Thread is analyzing key byte number 8
Thread is analyzing key byte number 9
Thread is analyzing key byte number 12
Thread is analyzing key byte number 7
Thread is analyzing key byte number 11
Thread is analyzing key byte number 13
Thread is analyzing key byte number 14
Thread is analyzing key byte number 15
Thread is analyzing key byte number 16
That took 314 milliseconds
Actual Key :  48 9D B4 B3 F3 17 29 61 CC 2B CB 4E D2 E2 8E B7
Recovered Key : A8 17 F9 A3 A4 0C 04 02 C1 71 13 8A 54 D8 62 55
Recovered number bits:70

Retrieving secret key using 60 number of traces...
Thread is analyzing key byte number 2
Thread is analyzing key byte number 3
Thread is analyzing key byte number 6
Thread is analyzing key byte number 1
Thread is analyzing key byte number 4
Thread is analyzing key byte number 5
Thread is analyzing key byte number 8
Thread is analyzing key byte number 7
Thread is analyzing key byte number 9
Thread is analyzing key byte number 10
Thread is analyzing key byte number 14
Thread is analyzing key byte number 11
Thread is analyzing key byte number 12
Thread is analyzing key byte number 16
Thread is analyzing key byte number 15
Thread is analyzing key byte number 13
That took 983 milliseconds
Actual Key :  48 9D B4 B3 F3 17 29 61 CC 2B CB 4E D2 E2 8E B7
Recovered Key : 48 9D B4 33 F3 17 23 61 1D 2B CB 4E D2 E2 7D B7
Recovered number bits:115
```

Figure 2.1.8 Sample output from Java Console

Data Visualization

The graph representation of the correlation coefficient will be done using Python script.We will be utilizing the matplotlib.pyplot and pandas library to aid us with the visualization.

Highlighting the Max line

To the highlight maximum line, the data of all the graphs is plotted in the grey, including the highest graph. The data representation that is used to plot this graph is based on a table, whereby the data of a graph is in a column. Next, the maximum of data in the last row is extracted. This extract the last maximum data among all the graphs. Next, we extract the column number that contains that maximum data. Lastly, we get the graph based on the column number and change its properties to make it thicker and red in colour.

Additionally, after the graph has been plotted, an image of the graph will be saved in the local directory. This method is coded as a function that uses the python pandas library and have different options for different outputs. This methods are only applied to dataset that will be plotted in correlation against number of traces recovered graph.

```
def plot_graph(file_name, axis=None, transpose=True, last_data=False):
  offset=1;
  data = pd.read_csv(file_name, header=None,index_col=0)
  largest_len = data.index.tolist()[-1]

  if transpose:
    data = data.T

  #take the last data for comparision, the graph may settle to a steady value at the end of the graph
  if last_data:
    max_index= data.columns[(data.iloc[-1] == data.iloc[-1].max())].tolist()[0] -offset
    print (hex(int(max_index/2)))

  else:
    max_index= data.columns[(data == data.max())].tolist()[0] -offset
    print (hex(int(max_index/2)))

  if axis != None:
    data.plot(color='grey',legend=None, ax=axis)
    axis.get_lines()[max_index].set(color='red', linewidth=1.5,zorder=1000)
    #axis.get_lines()[max_index].set_color('red')
    axis.set_xlabel("Number of Traces")
    axis.set_ylabel("Correlation")

    axis.set_title("Key : " + str(hex(int(max_index/2))))

    plt.tight_layout()

    fig = axis.get_figure()
    fig.set_size_inches(18.5, 10.5)
    fig.savefig("Graphs.png")
  else:
    plot = data.plot(color='grey',legend=None)
    plot.get_lines()[max_index].set(color='red', linewidth=1.5,zorder=1000)
    #plot.get_lines()[max_index].set_color('red')
    plot.set_xlabel("Number of Traces")
    plot.set_ylabel("Correlation")

    plt.text(largest_len-20, data.iloc[-1,max_index]+0.1, "Key:"+str(hex(int(max_index/2))), fontdict=None, withdash=False)

    fig = plot.get_figure()
    fig.savefig("plot_"+file_name[:-4]+".png")
```

To plot the graph of number of traces against bits recovered, a csv file that contains this data is read. Using the same panda library in python, the graph is plotted based on the labels and title. After plotting, an image is saved in the local directory. The picture is seen in Figure 2.2.2.

```
data = pd.read_csv("Graph.csv", header=None,index_col=1,usecols=[0,1])
plot = data.plot(legend=None)

plot.set_title("Graph of number of traces againts number of bits recovered")

plot.set_ylabel("Number of Traces")
plot.set_xlabel("Number of bits recovered")

fig = plot.get_figure()
fig.savefig("traces vs bits recovered.png")
```

Sample output

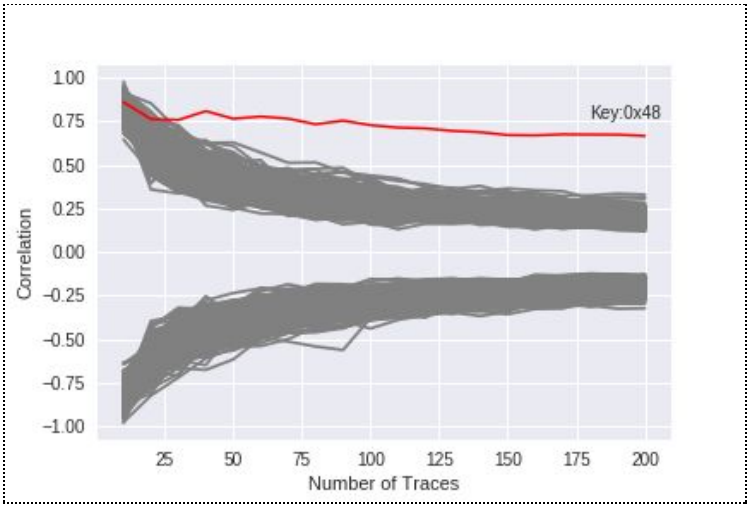

Figure 2.2.1 Graph of Correlation Coefficient vs No. of traces for the 1st byte of key

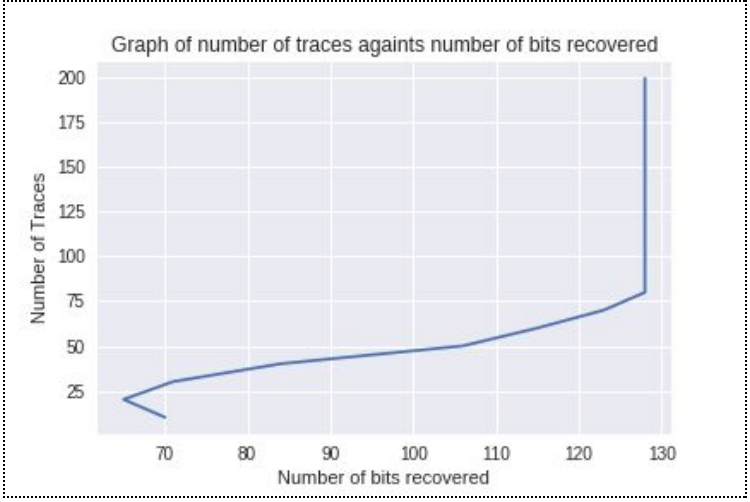

Figure 2.2.2 Graph of No. of traces vs No. of bits recovered

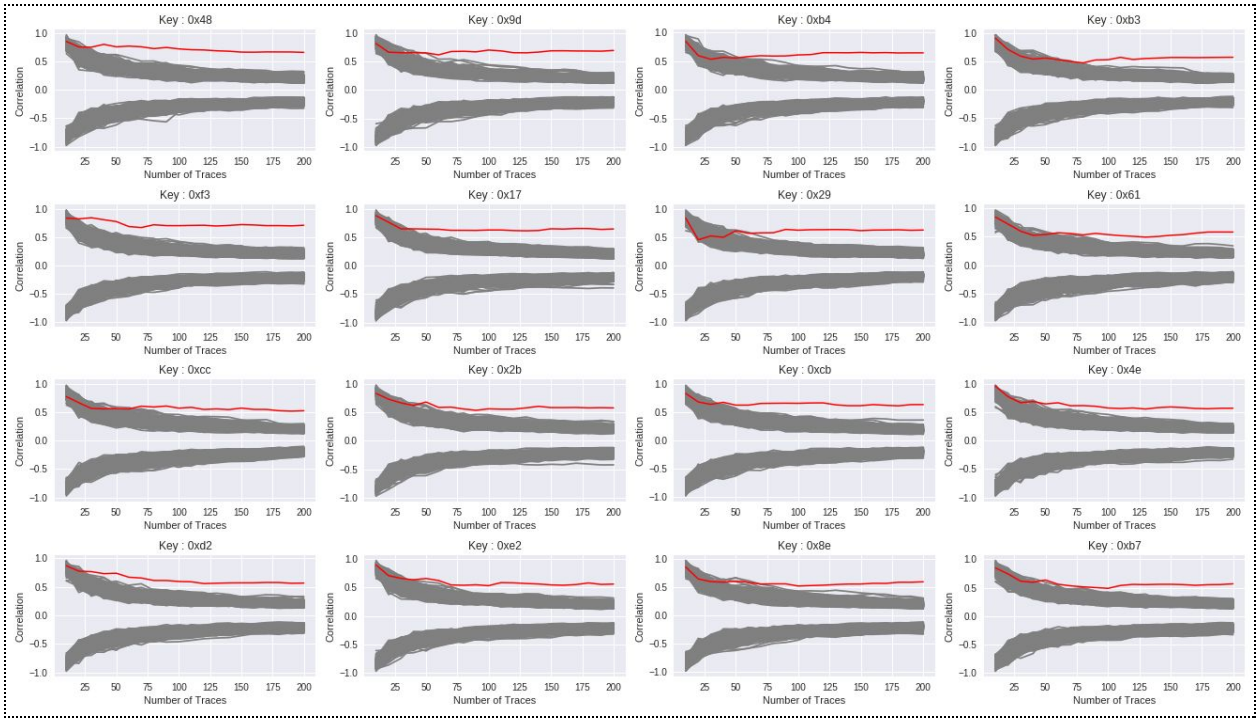

Figure 2.2.3 Graphs of Correlation Coefficient vs No. of traces
for the all bytes of key

3. Results and Evaluation

Correlation Coefficient vs Hypothesis (for 100 traces)

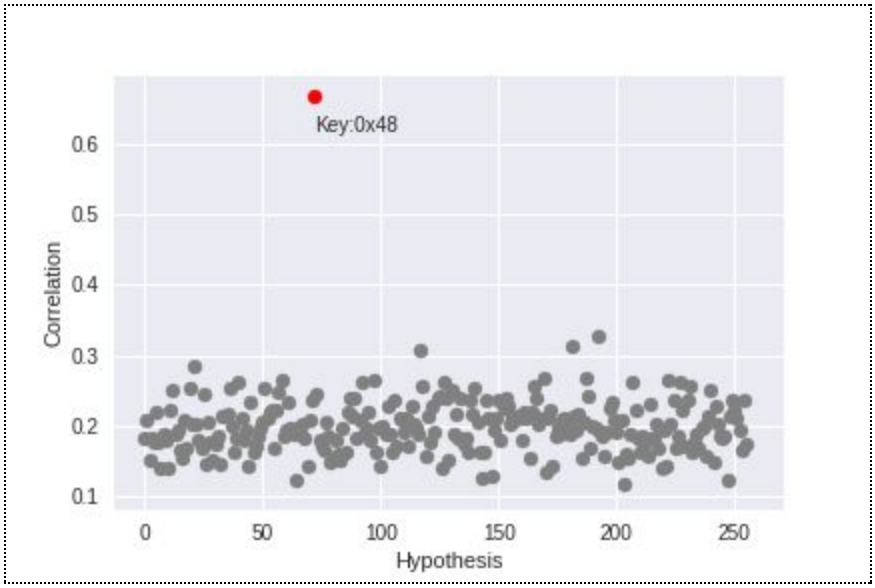

Figure 3.1.1 Graph of Correlation Coefficient vs Hypothesis
for the 1st byte of key

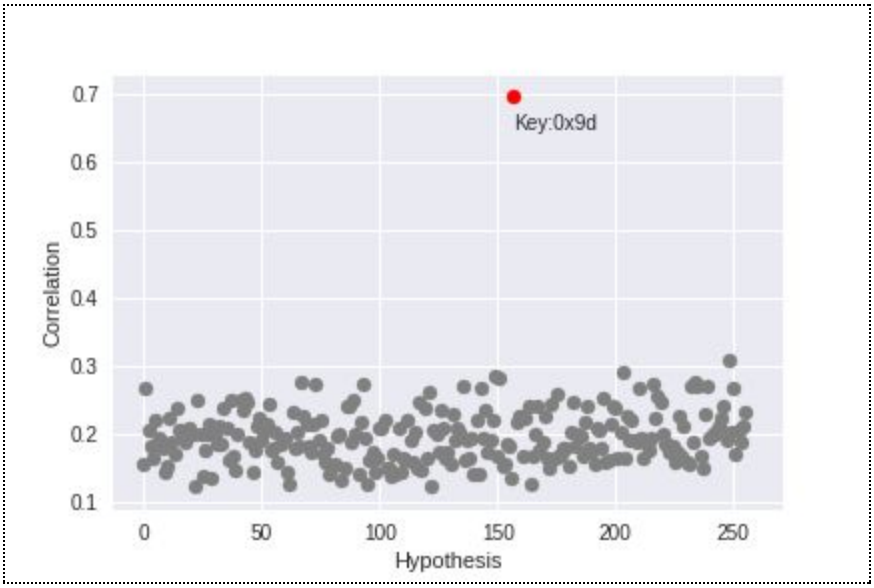

Figure 3.1.2 Graph of Correlation Coefficient vs Hypothesis for the 2nd byte of key

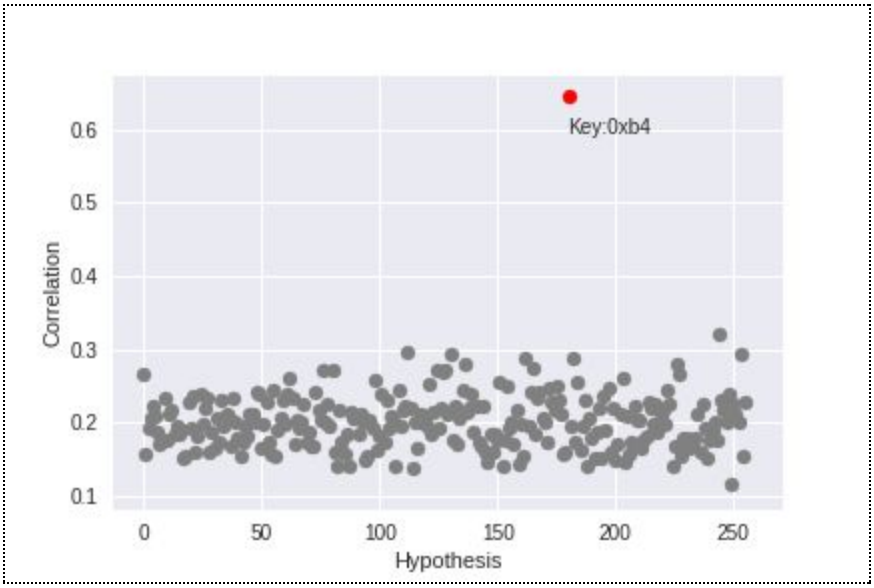

Figure 3.1.3 Graph of Correlation Coefficient vs Hypothesis for the 1st byte of key

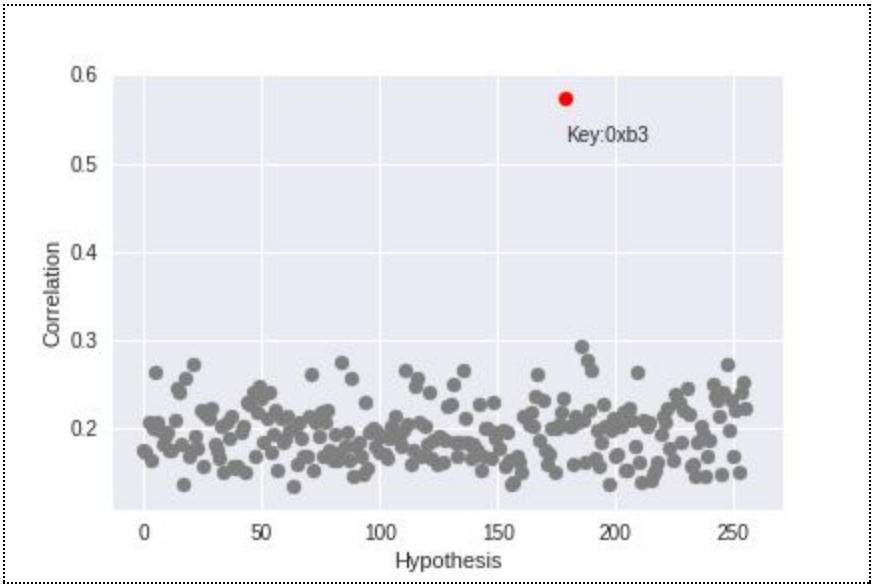

Figure 3.1.4 Graph of Correlation Coefficient vs Hypothesis for the 4th byte of key

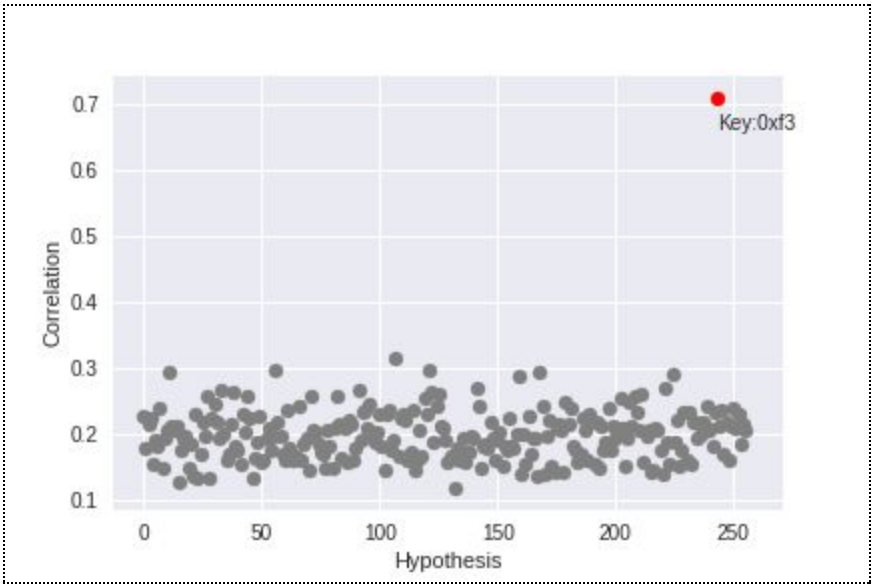

Figure 3.1.5 Graph of Correlation Coefficient vs Hypothesis for the 5th byte of key

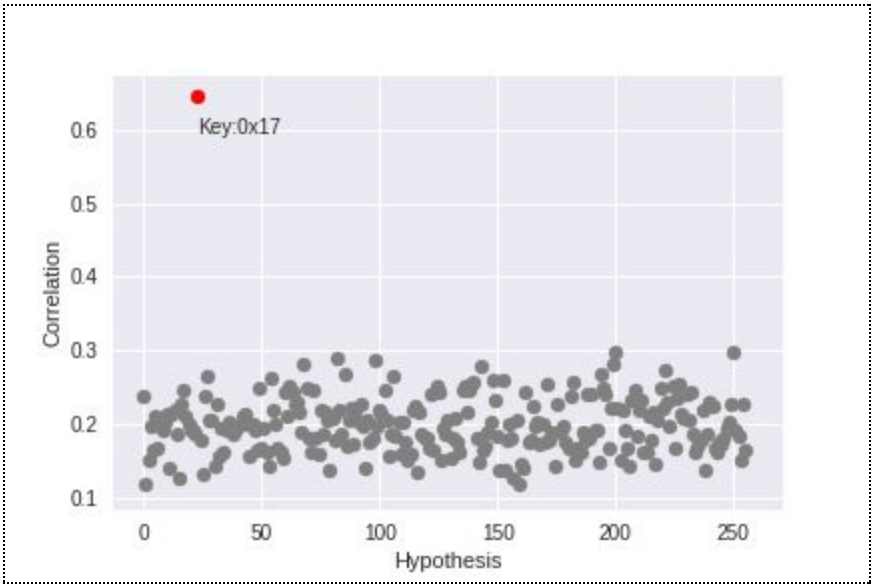

Figure 3.1.6 Graph of Correlation Coefficient vs Hypothesis for the 6th byte of key

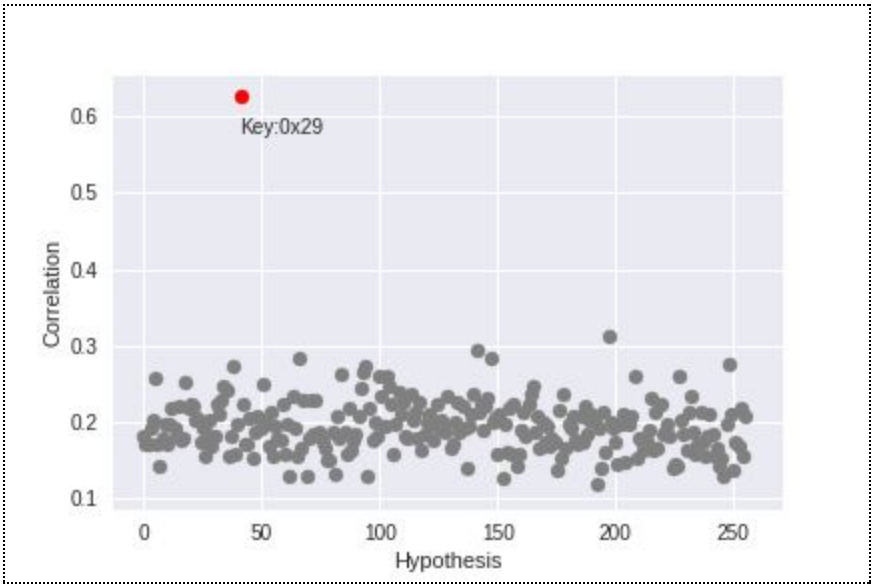

Figure 3.1.7 Graph of Correlation Coefficient vs Hypothesis for the 7th byte of key

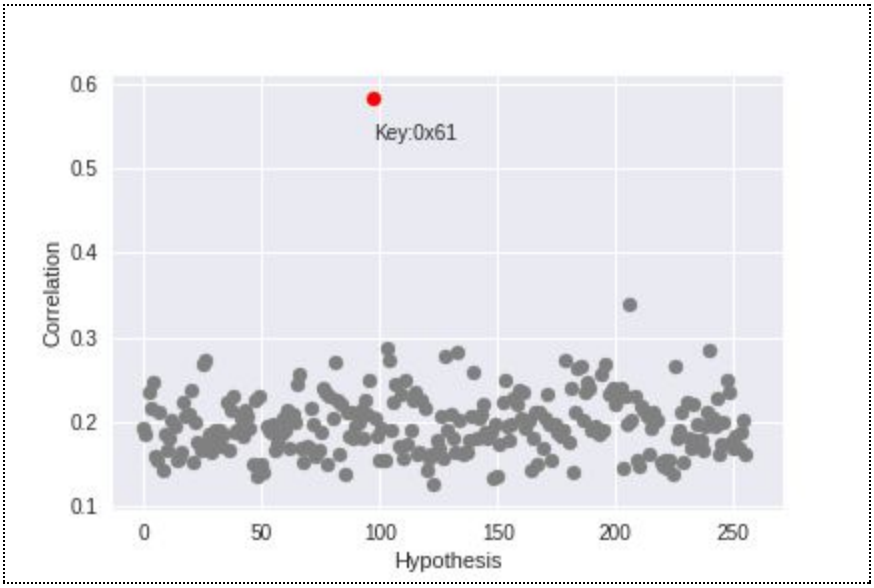

Figure 3.1.8 Graph of Correlation Coefficient vs Hypothesis for the 8th byte of key

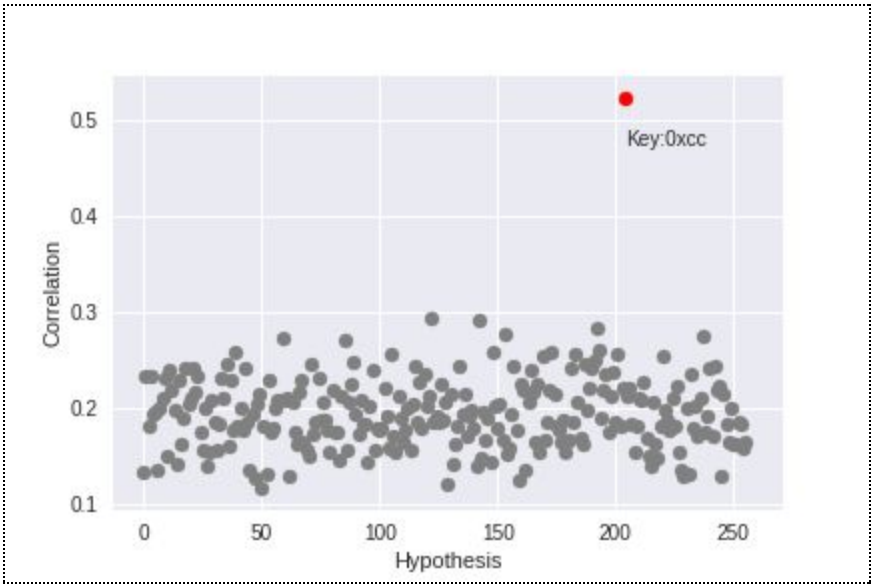

Figure 3.1.9 Graph of Correlation Coefficient vs Hypothesis for the 9th byte of key

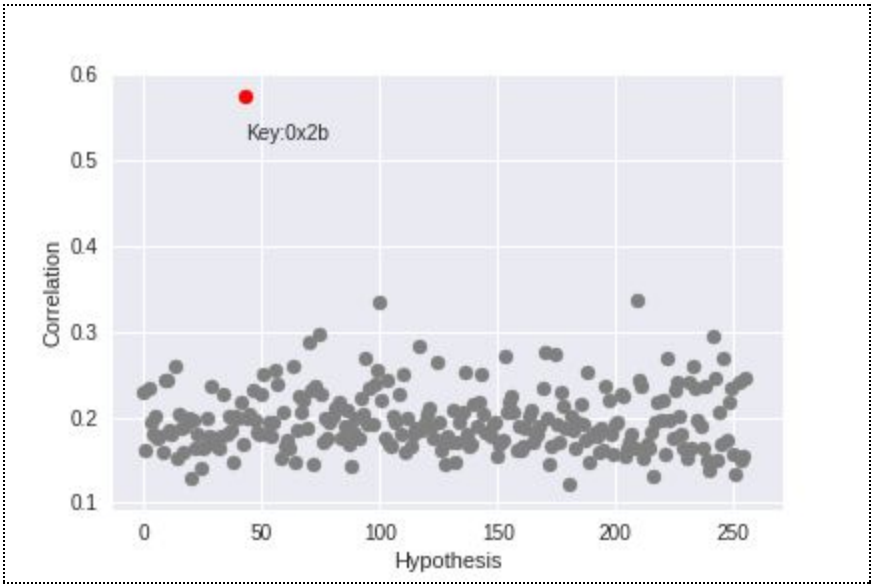

Figure 3.1.10 Graph of Correlation Coefficient vs Hypothesis for the 10th byte of key

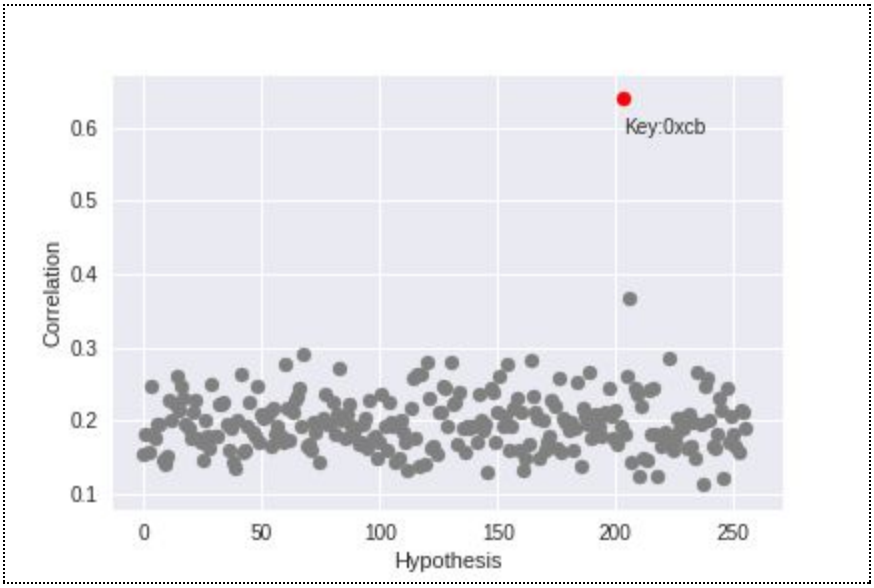

Figure 3.1.11 Graph of Correlation Coefficient vs Hypothesis for the 11th byte of key

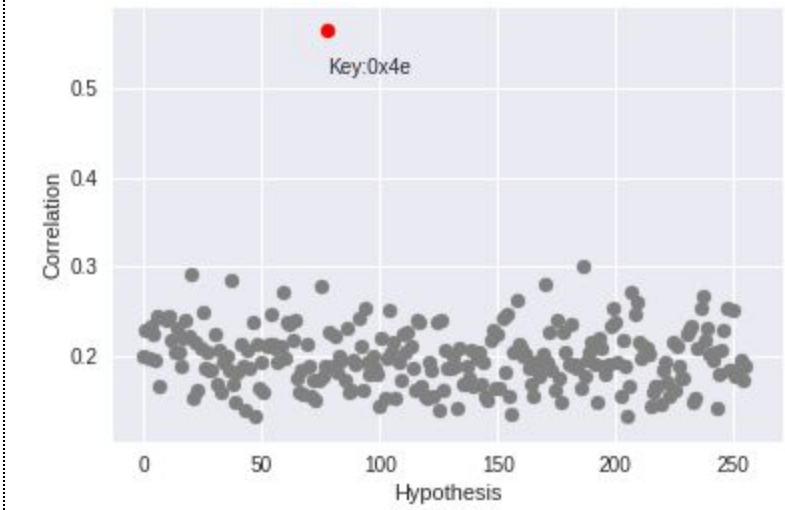

Figure 3.1.12 Graph of Correlation Coefficient vs Hypothesis for the 12th byte of key

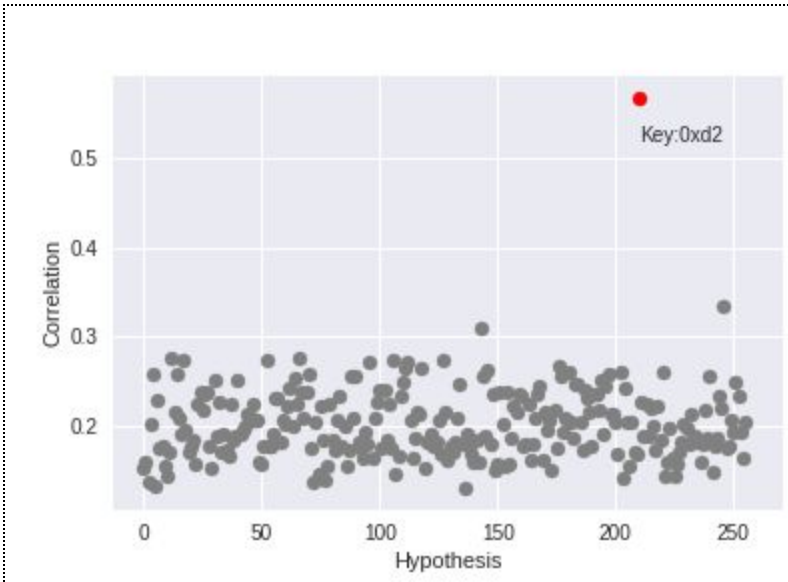

Figure 3.1.13 Graph of Correlation Coefficient vs Hypothesis for the 13th byte of key

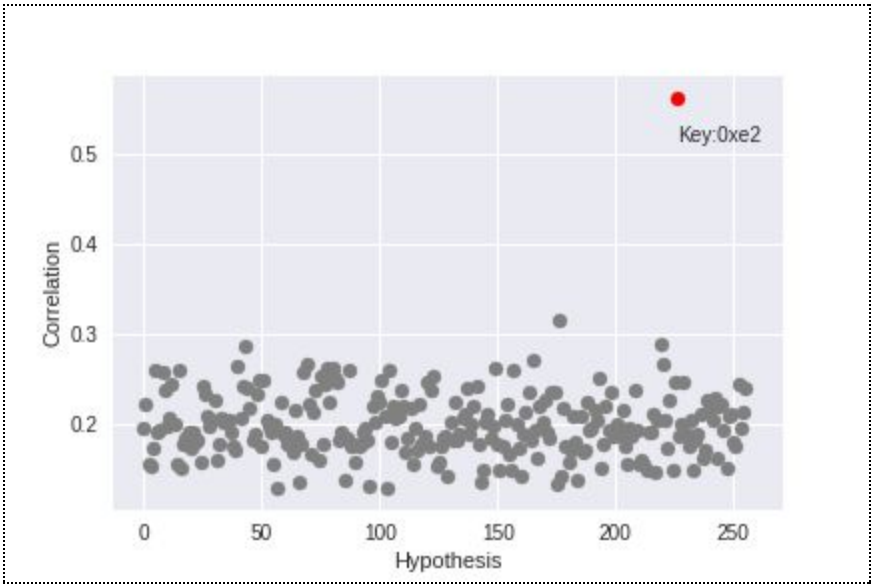

Figure 3.1.14 Graph of Correlation Coefficient vs Hypothesis for the 14th byte of key

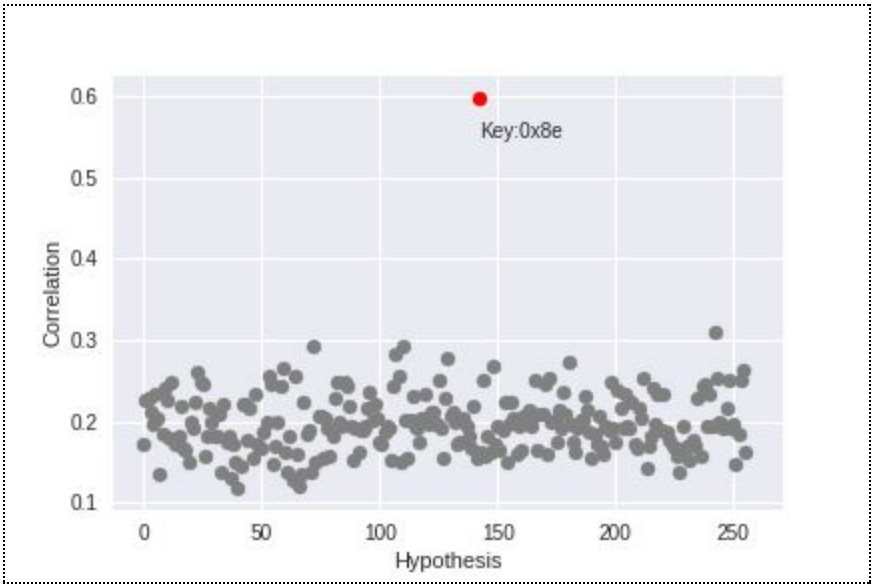

Figure 3.1.15 Graph of Correlation Coefficient vs Hypothesis for the 15th byte of key

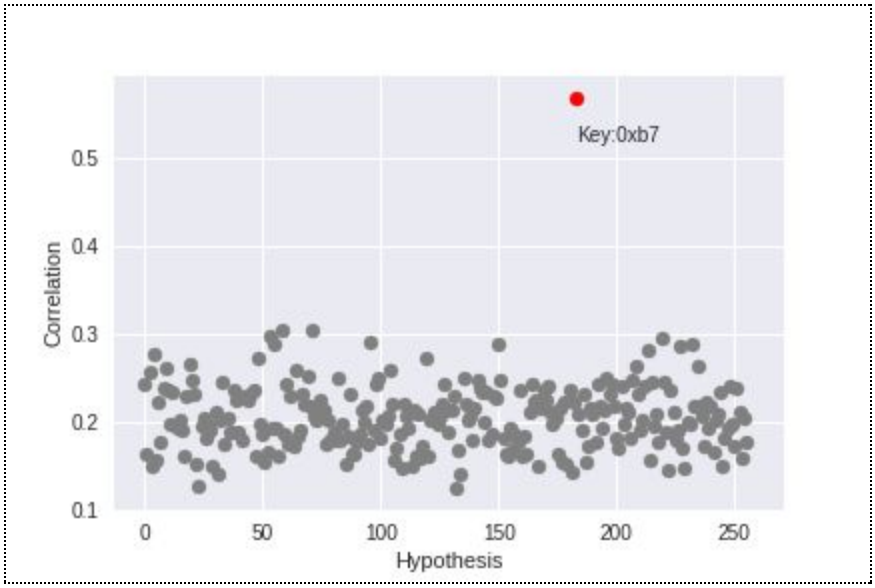

Figure 3.1.16 Graph of Correlation Coefficient vs Hypothesis
for the 16th byte of key

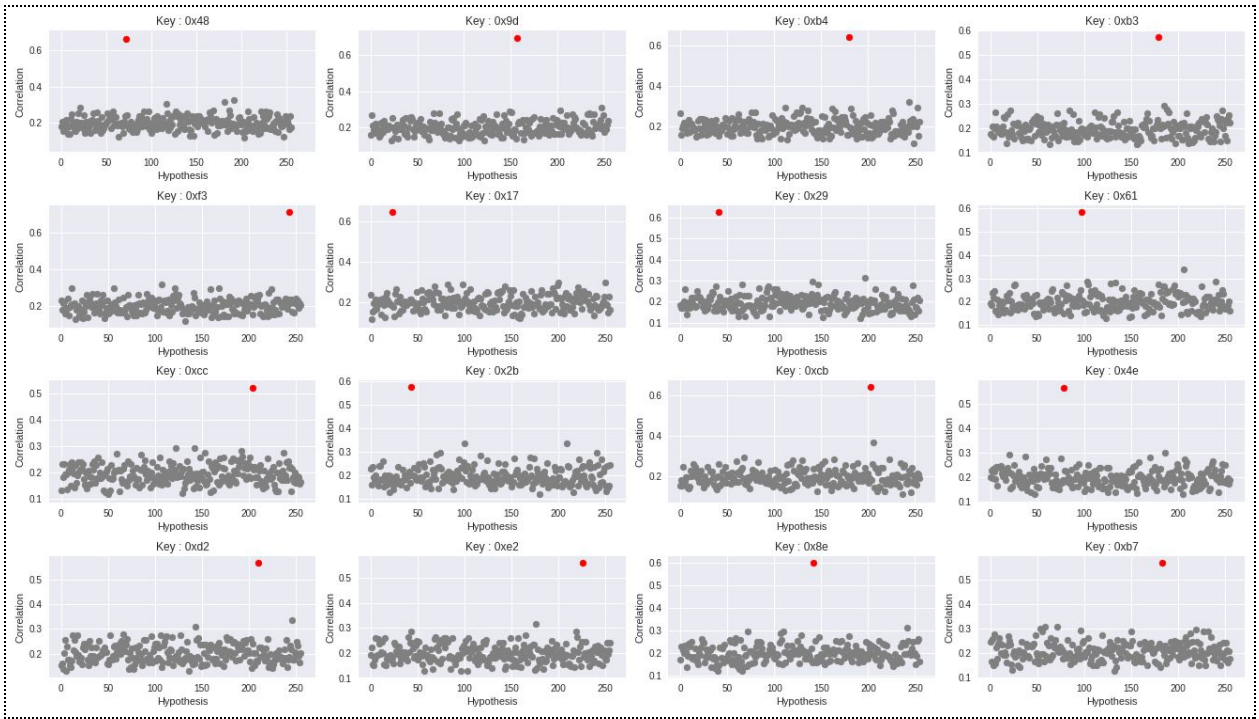

Figure 3.1.17 Graphs of Correlation Coefficient vs Hypothesis (100 traces)
for the all bytes of key

Graph of Correlation of Correct Key Byte vs Number of Traces

Our group has decided to log down the **maximum** and **minimum** correlation value to show that as the number of traces increases, the correlation coefficient will eventually converge and only the correct key(highlighted in red) will be the one that is not converging.

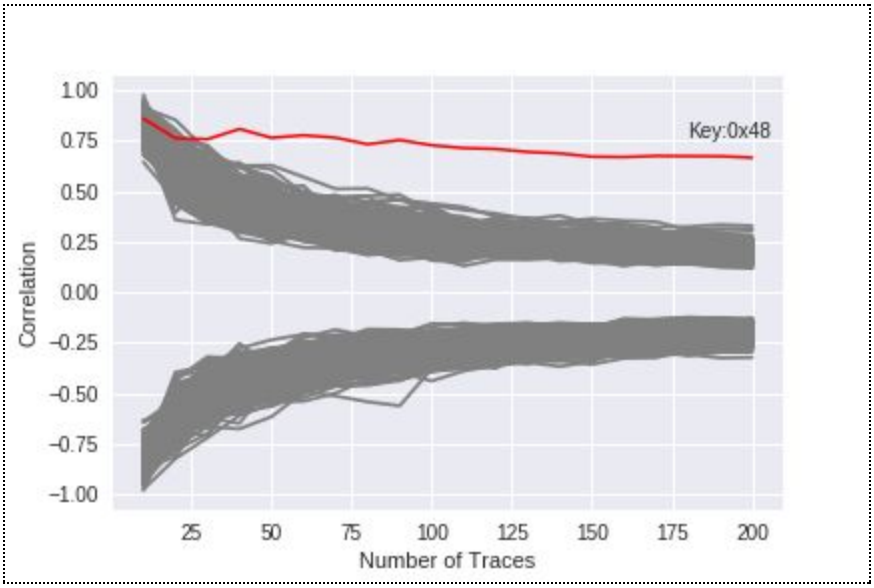

Figure 3.2.1 Graph of Correlation Coefficient vs No. of traces for the 1st byte of key

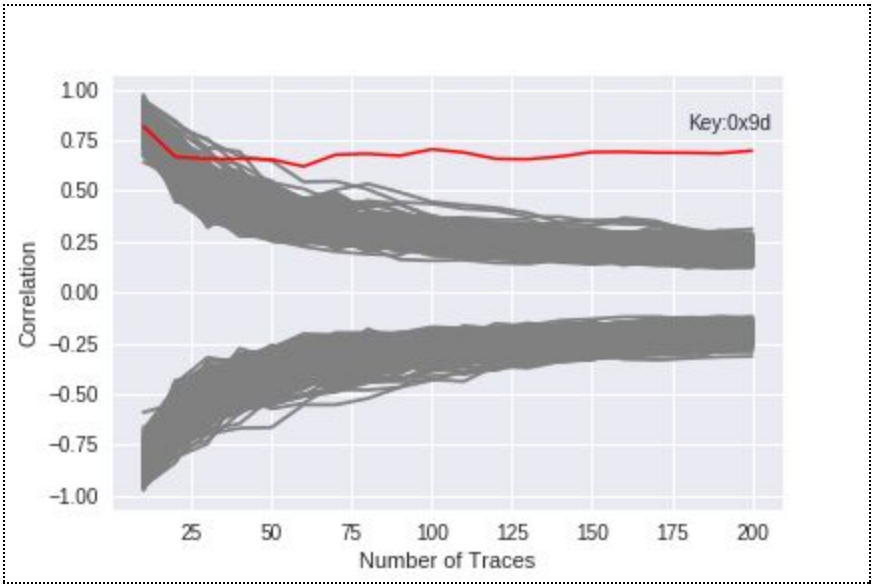

Figure 3.2.2 Graph of Correlation Coefficient vs No. of traces for the 2nd byte of key

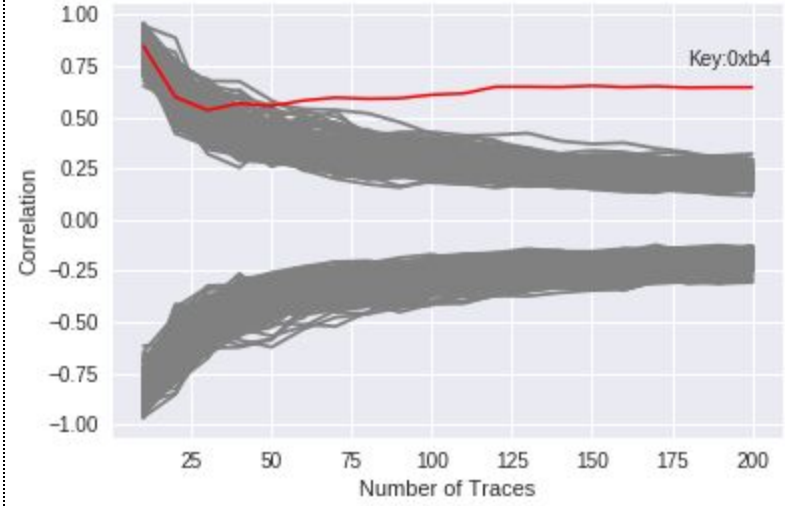

Figure 3.2.3 Graph of Correlation Coefficient vs No. of traces for the 3rd byte of key

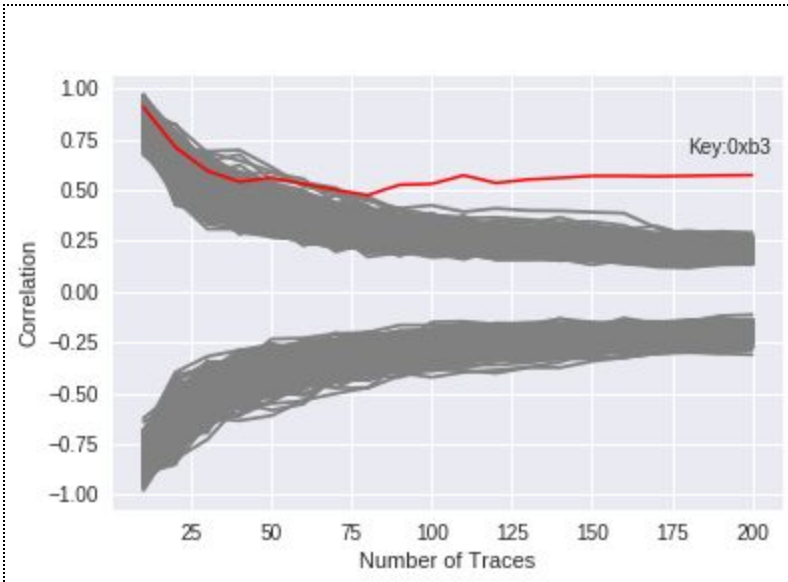

Figure 3.2.4 Graph of Correlation Coefficient vs No. of traces for the 4th byte of key

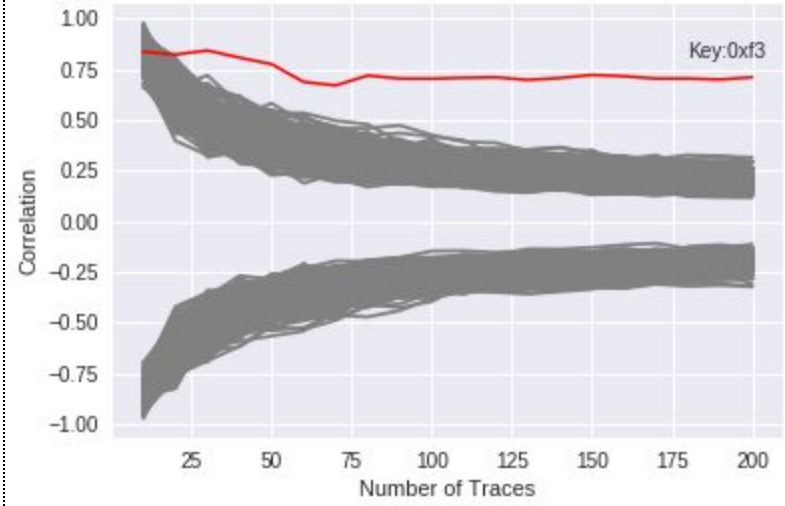

Figure 3.2.5 Graph of Correlation Coefficient vs No. of traces for the 5th byte of key

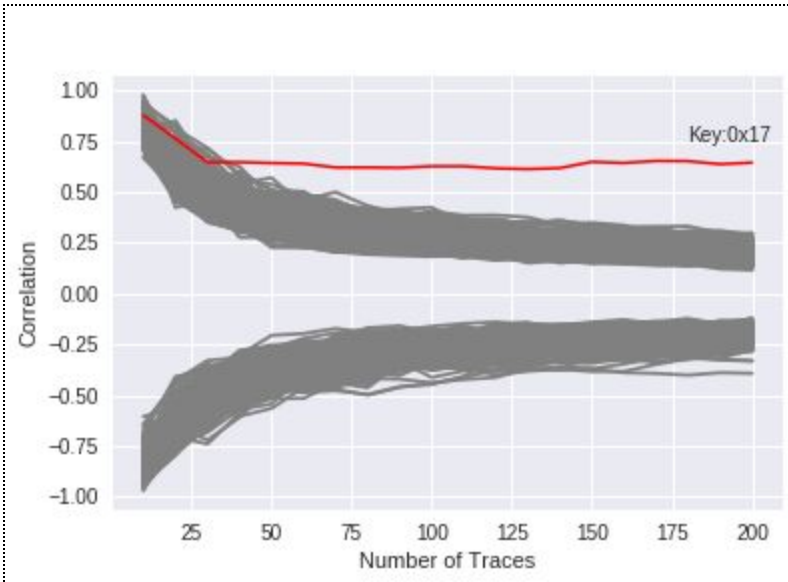

Figure 3.2.6 Graph of Correlation Coefficient vs No. of traces for the 6th byte of key

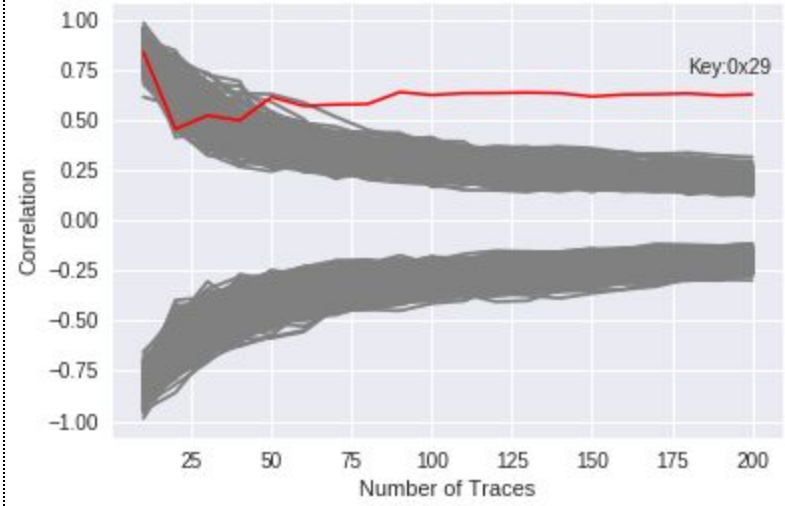

Figure 3.2.7 Graph of Correlation Coefficient vs No. of traces for the 7th byte of key

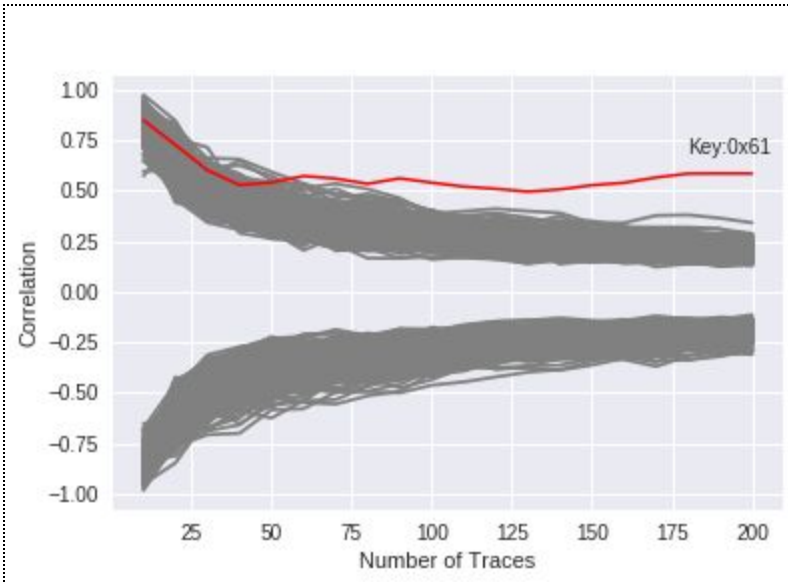

Figure 3.2.8 Graph of Correlation Coefficient vs No. of traces for the 8th byte of key

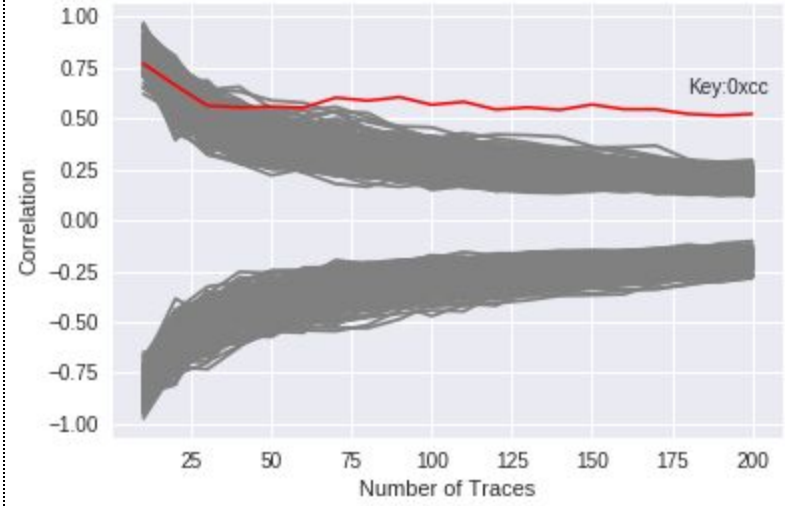

Figure 3.2.9 Graph of Correlation Coefficient vs No. of traces for the 9th byte of key

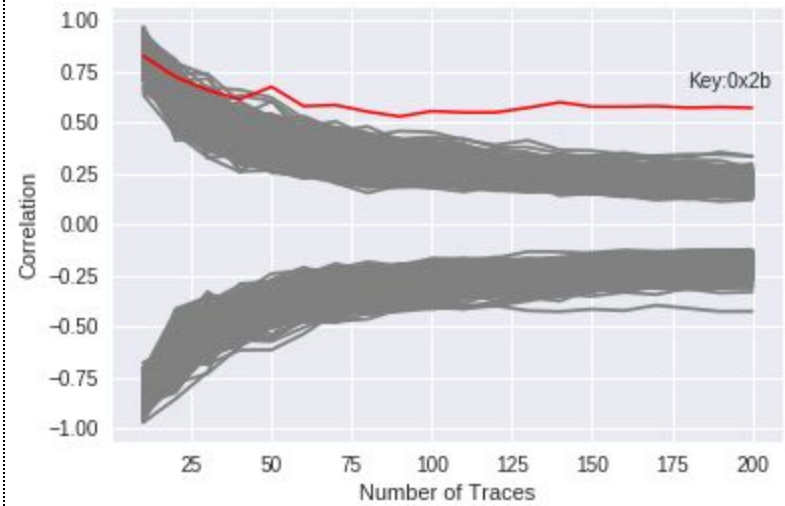

Figure 3.2.10 Graph of Correlation Coefficient vs No. of traces for the 10th byte of key

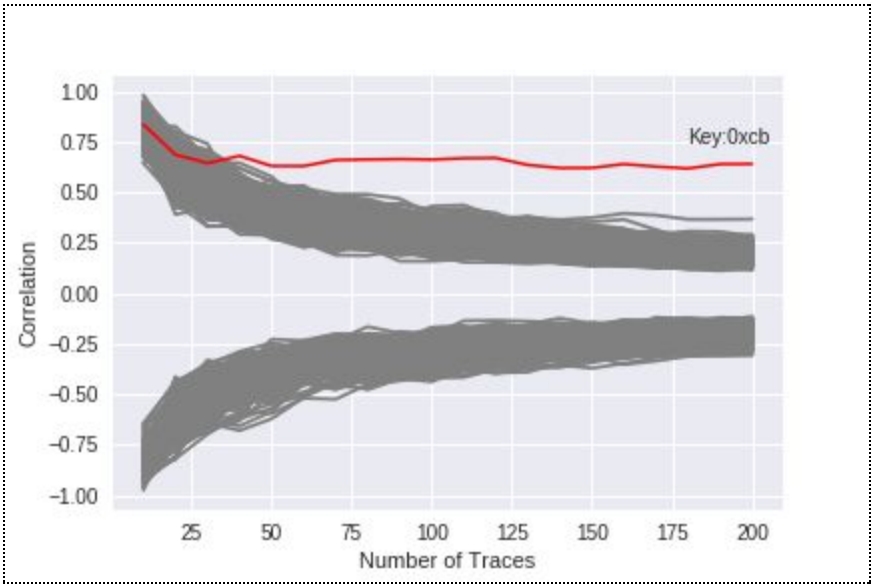

Figure 3.2.11 Graph of Correlation Coefficient vs No. of traces for the 11th byte of key

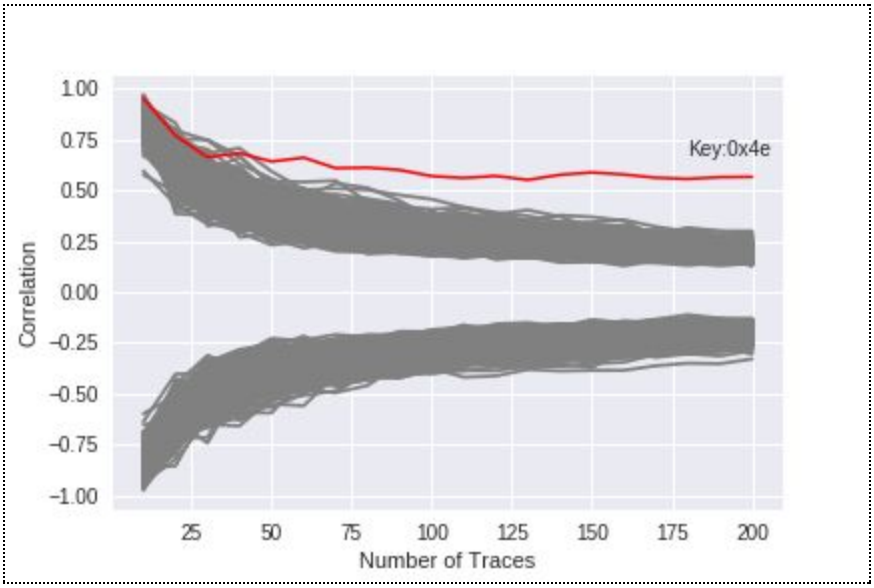

Figure 3.2.12 Graph of Correlation Coefficient vs No. of traces for the 12th byte of key

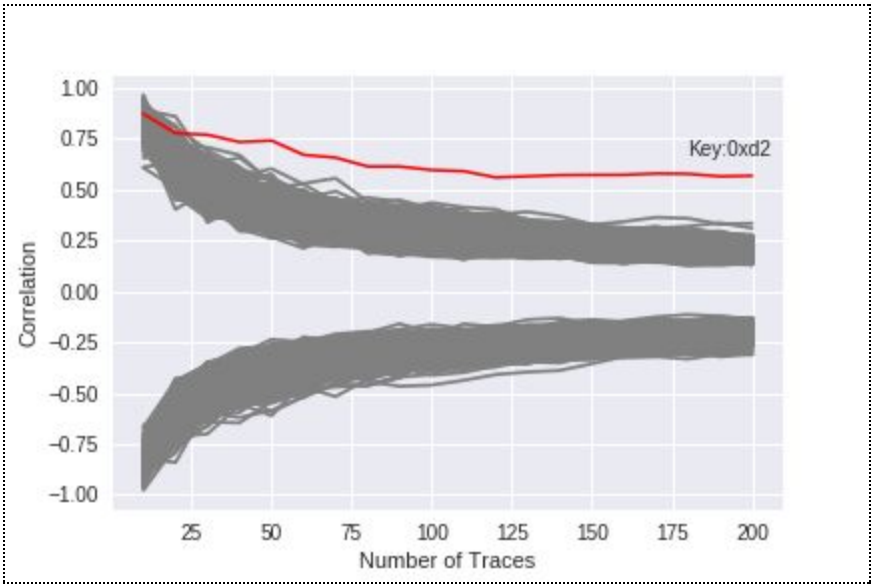

Figure 3.2.13 Graph of Correlation Coefficient vs No. of traces for the 13th byte of key

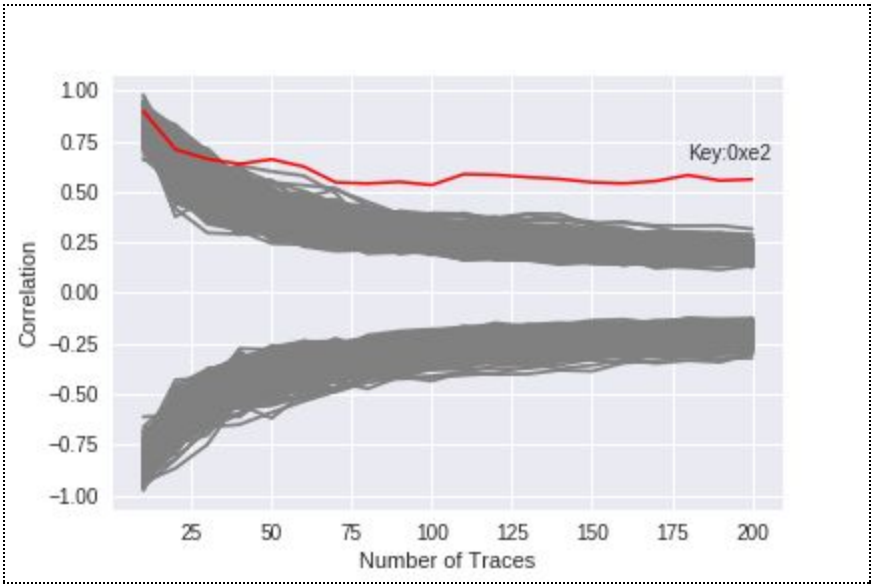

Figure 3.2.14 Graph of Correlation Coefficient vs No. of traces for the 14th byte of key

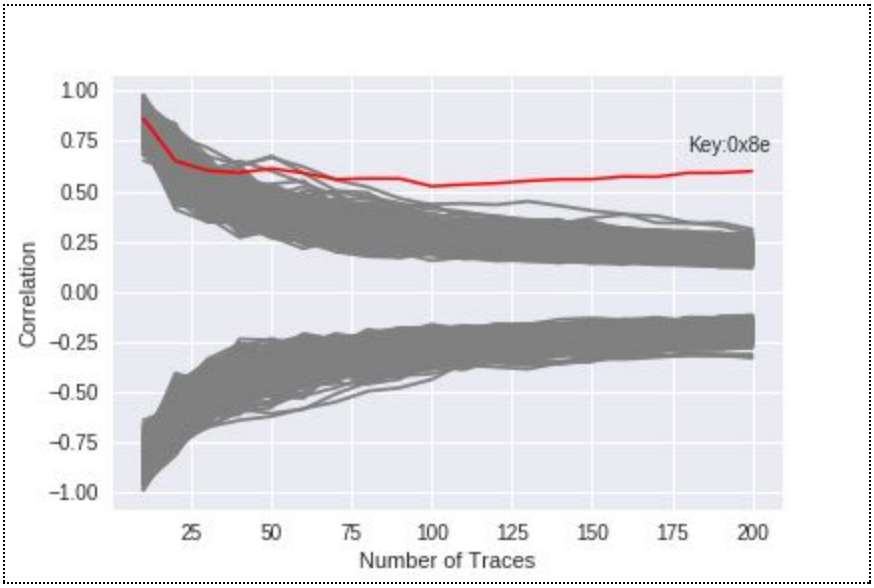

Figure 3.2.15 Graph of Correlation Coefficient vs No. of traces for the 15th byte of key

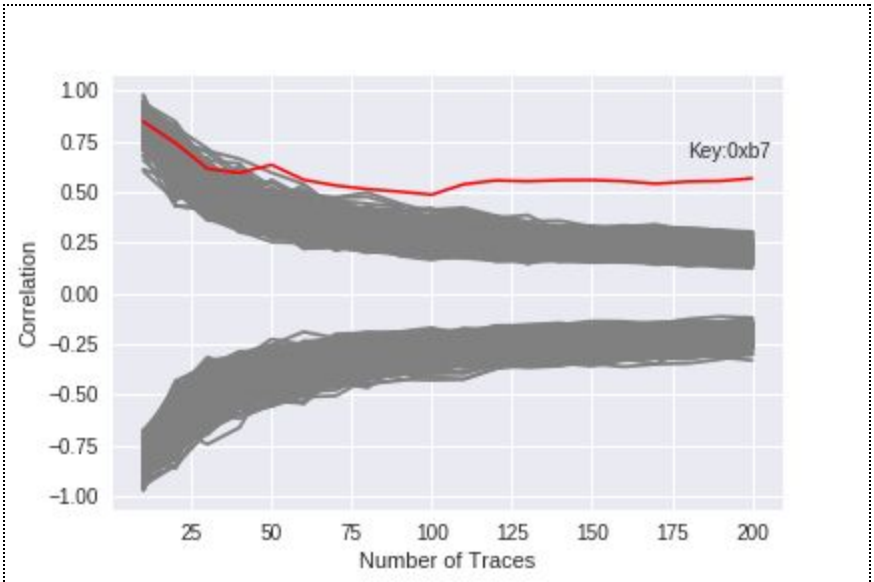

Figure 3.2.16 Graph of Correlation Coefficient vs No. of traces for the 16th byte of key

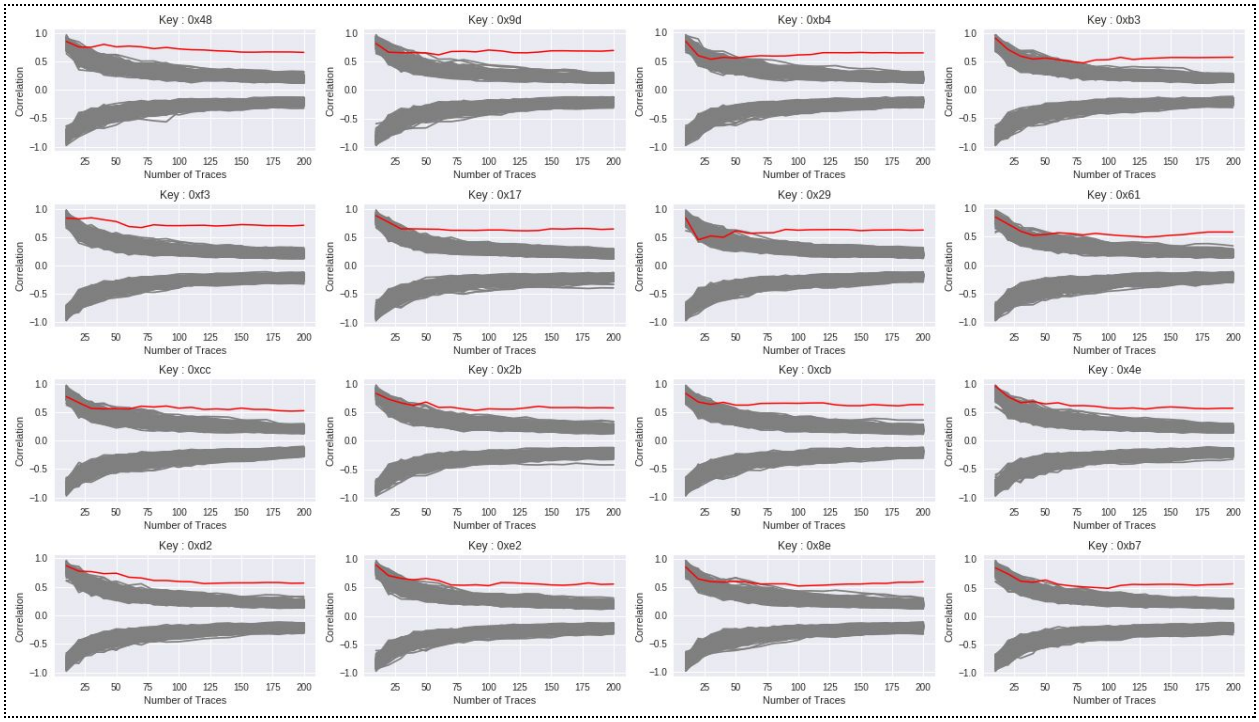

Figure 3.2.17 Graphs of Correlation Coefficient vs No. of traces for the all bytes of key

No. of Traces vs No. of Bits Recovered

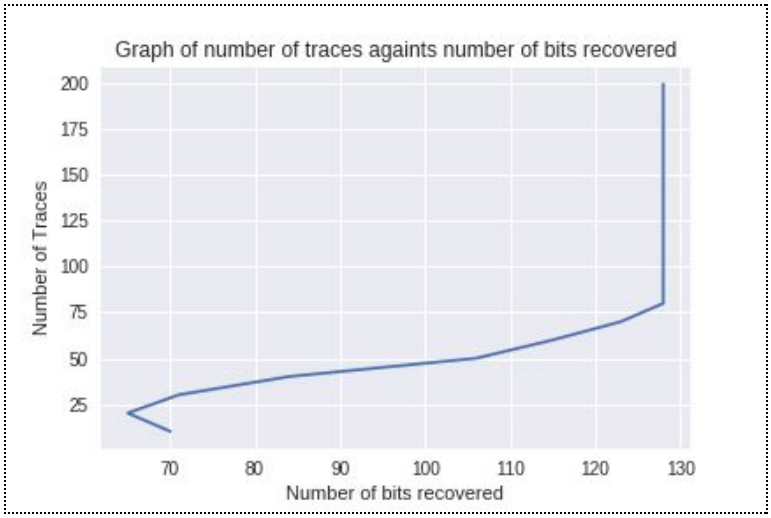

Figure 3.2.18 Graph of No. of traces vs No. of bits recovered

Comparison between lab's PAT software

In order to verify that all our implementations are correct, a comparison between our results and the lab's PAT tool will be done.

The experiment is conducted in Software Project Lab 1 (N4-01a-02) and below are the results of the comparison.

Figure 3.3.1, Figure 3.3.2, Figure 3.3.3, Figure 3.3.4 and Figure 3.3.5 are the screenshots of the comparisons of outputs with varying number of traces from 10, 60, 100, 300 to 1500 respectively. Number of bits recovered should be 128 - sum(unmatched bits) from the PAT tool.

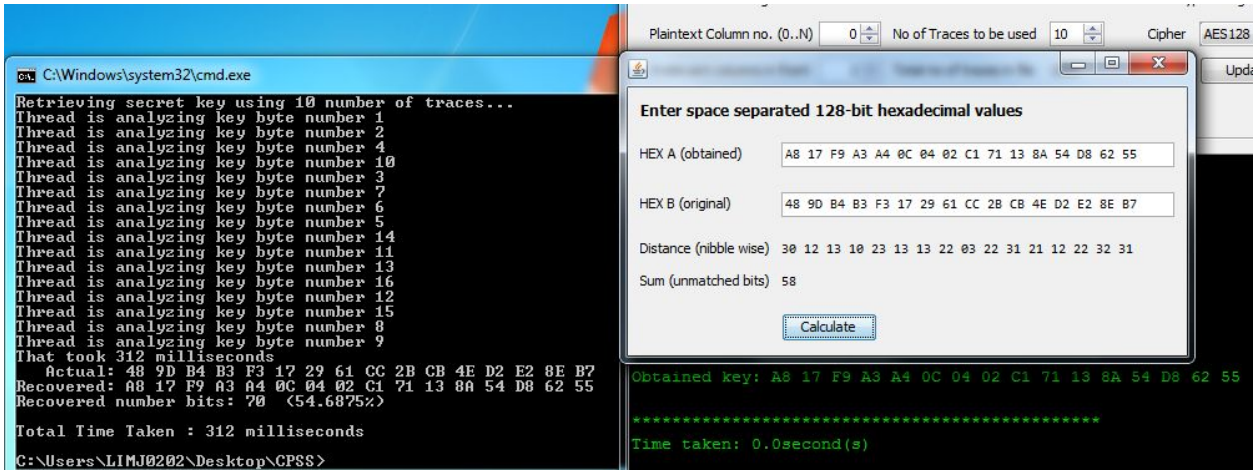

Figure 3.3.1 Comparison between outputs with 10 traces

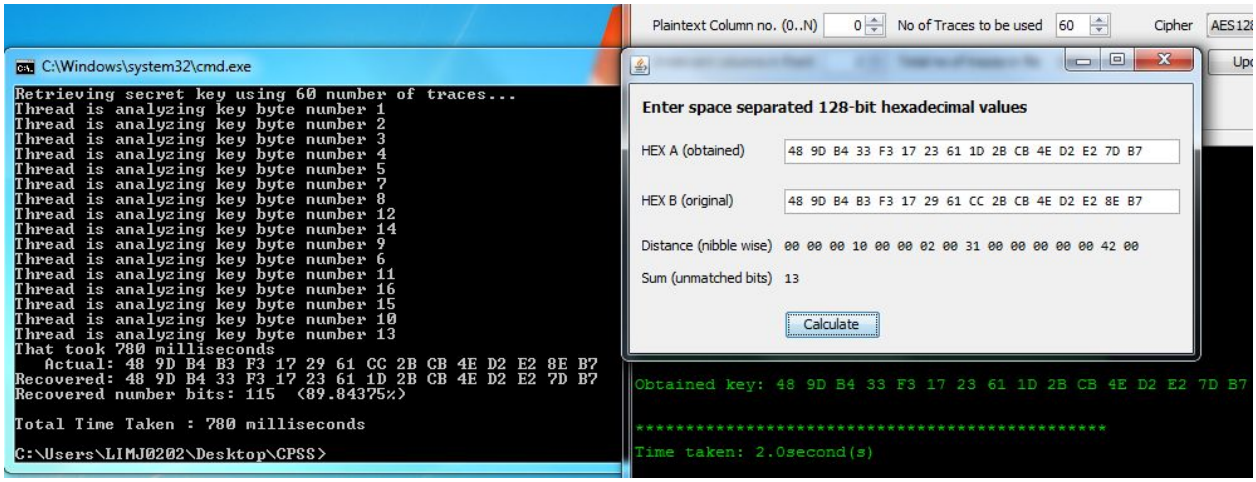

Figure 3.3.2 Comparison between outputs with 60 traces

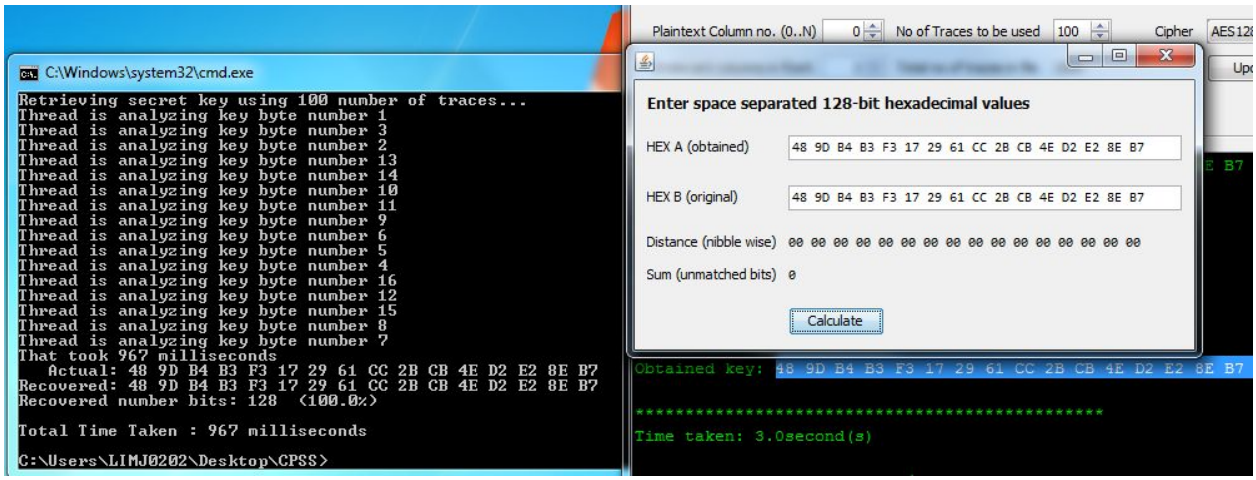

Figure 3.3.3 Comparison between outputs with 100 traces

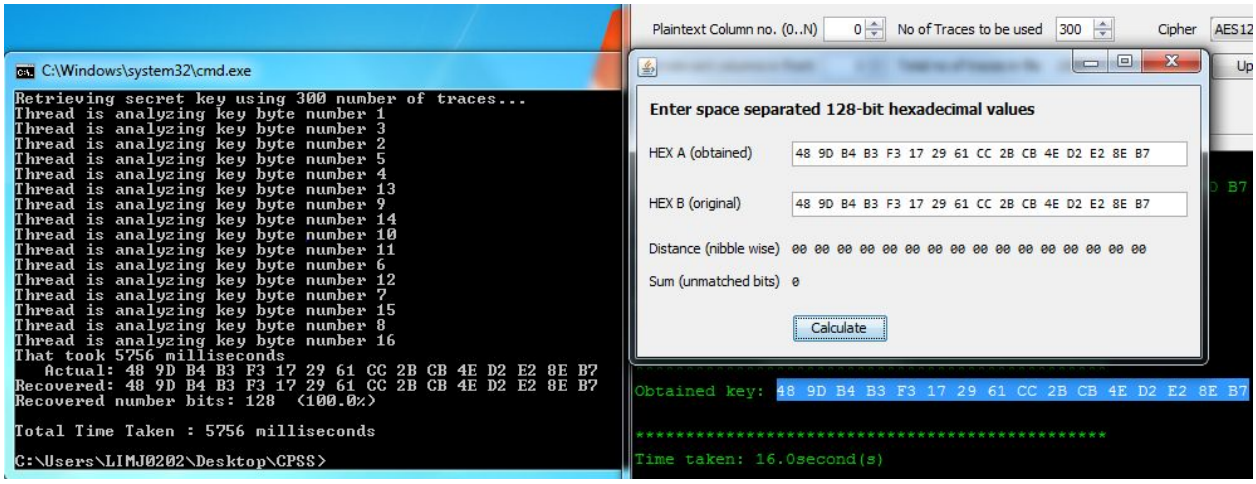

Figure 3.3.4 Comparison between outputs with 300 traces

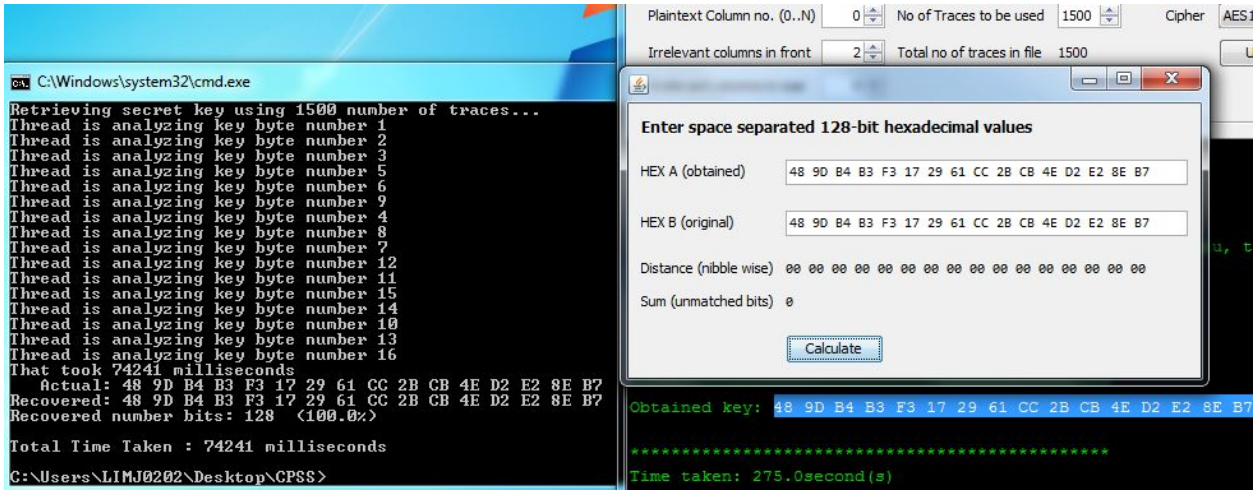

Figure 3.3.5 Comparison between outputs with 1500 traces

From the results, we can verify that the implementation of our secret key retrieval as well as calculating the number of bits recovered is correct.

We can also see that our group's implementation of recovering the key through multithreading takes around 3 times lesser computational time when computing for large sample trace collections. Computation time is important as sometimes, the trace collection will go up to thousands.

4. Future Works

Changing of programming language

Since the correlation calculation relies heavily on arithmetic operations, we might want to consider switching to C++ as the performance for arithmetic operations is faster than Java.

In addition, we might also want to write the C++ application in a multithreading manner to speed up even further for large amount of trace analysis.

Thread Management

The CPA application is currently written in a way that where it will start *x* threads where *x* is the number of key bytes.

It is typically not recommended to start number of threads that is greater than number of available processors. If the number of threads started is greater than number of available processors, it will only introduce overheads for context switching and memory cache flushing.

We might want to figure out how many available processors the computer has and start the correct number of threads at a single time.

We can also implement pipeline architecture into the application to do parallel processing of data.

5. Countermeasures against Power Analysis Attacks

Countermeasures can be performed on both hardware and software level. When a device performs cryptographic operations, information such as the signal and noise will be leaked. Signal is the information that is correlated useful to the secret key, while noise is the information that hinder the determination of the secret key. The quality of the information gained from power traces is characterized by a “signal to noise” (S/N) ratio, which is a measurement of signal compared to the amount of noise. One example of hardware level countermeasures is introducing noise to the power line. When noise is manually added using some noise generating device, it decreases the S/N ratio. Therefore, the number of power traces needed to perform a successful attack increases as the number of power traces needed is inversely proportional to the square of the S/N ratio. As a result, in order to make the attack work harder, the signal-to-noise ratio should be reduced to the lowest possible level.

Another example of hardware level countermeasure is the clock skipping technique, which is also known as clock decorrelation. It decorrelates cryptographic operations from normal external clock cycles by using an internally generated clock signal. The internal clock is used only to control the timing of the processor during the cryptographic operations. All I/O communication operations are controlled by an external clock to preserve the correctness of communication between the device and the external reader. This technique makes the alignment of data points in the collected power traces more difficult and as a result it reduces the S/N ratio.

There are many types of countermeasures that can be performed on the software level. Dummy instructions such as NOP (No Operation) can be randomly inserted into the encryption code. This breaks the alignment of power traces and hence more power traces need to be obtained in order to perform a successful attack. The con of this countermeasure is that the power consumption for NOP instructions are easily distinguishable. Therefore, power traces can be visually inspected to detect the NOP instructions through techniques such as Simple Power Analysis. Hence, this means that Simple Power Analysis attack cannot be prevented using this method.

Another example of software level countermeasure is to insert real instructions into the code. This method is similar to the insertion of dummy instructions that is introduced above. The insertion of real instructions, however, makes it challenging to visually inspect the power traces. Hence, attacks such as Simple Power Analysis attack can be prevented using this method. Although this method results in a compromise in power consumption, it is easy to implement it on any microcontroller. Therefore this method is less costly and more feasible compared to other countermeasures.

6. Conclusion

Power analysis attacks usually cannot be detected by the affected device because the monitoring of the power consumption is passive and the attack is non-invasive. Therefore, countermeasures are constantly being improved by cryptosystem engineers to ensure that power analysis attacks can be prevented by making an attack difficult such that the reward of breaking the system is less than the cost of breaking it.

References

https://arxiv.org/pdf/1801.00932.pdf
https://patentimages.storage.googleapis.com/0c/ec/51/9c19a2f68c732a/US7599488.pdf
https://eprint.iacr.org/2005/022.pdf